合肥工業大學

操作系统实验报告

实验题目	实验 3 进程的创建
学生姓名	袁焕发
学　　号	2019217769
专业班级	物联网工程 19-2
指导教师	田卫东
完成日期	2021 年 11 月 2 日

合肥工业大学 计算机与信息学院

1. 实验目的和任务要求

依据操作系统课程所介绍的进程创建和运行过程，本实验的目的练习使用 EOS API 函数 CreateProcess 创建一个进程，掌握创建进程的方法，理解进程和程序的区别；调试跟踪 CreateProcess 函数的执行过程，了解进程的创建过程，理解进程是资源分配的基本单位；调试跟踪 CreateThread 函数的执行过程，了解线程的创建过程，理解线程是调度的基本单位，从而加深对教材上进程和线程概念的理解。

2. 实验原理

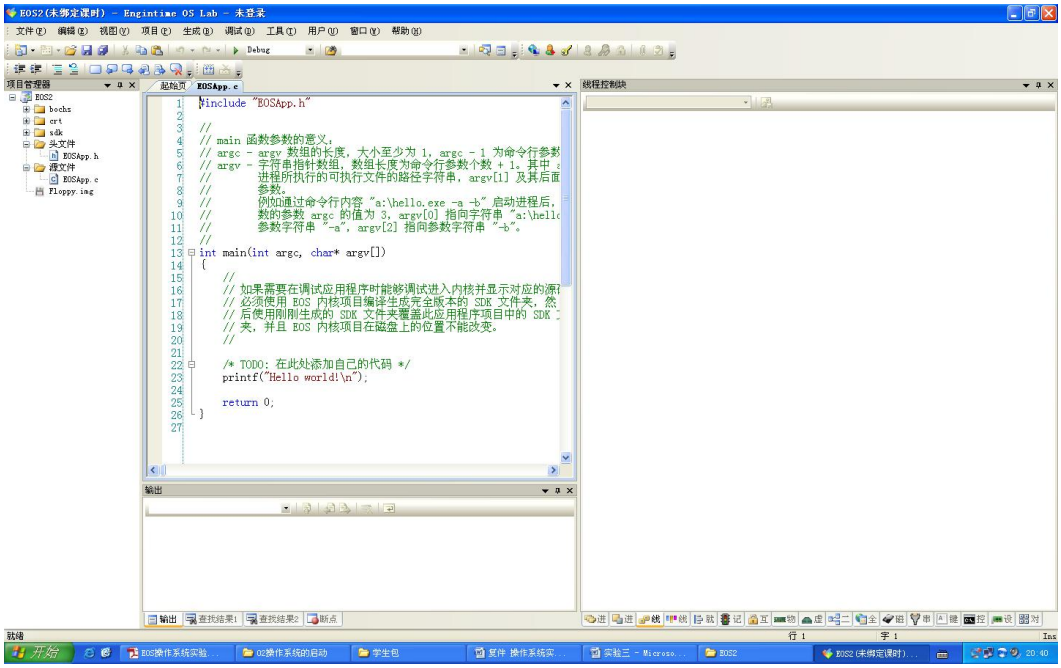
进程是具有独立功能的程序关于某个数据集合上的一次运行活动，是系统进行资源分配的单位。程序和进程是两个完全不同的概念，进程是动态的、是暂时的，而程序是静态的可以长久保存。一个程序通过多次执行，可以产生多个进程。一个进程通过调用不同的程序，可以包括多个程序。系统为了管理进程设置的一个专门的数据结构称为进程控制块(PCB)。

在 EOS 中，线程是处理器调度的基本单位。当一个进程被创建时，系统首先会为该进程分配一些资源，然后系统会为该进程创建一个默认线程，做为该进程的主线程。同样的为了管理线程有一个线程控制块（TCB）。在 EOS 内核中，线程的创建最终都是通过调用 PspCreateThread 函数，在 EOS 应用程序中可以调用 API 函数 CreateThread 来创建一个新线程。

3. 实验内容

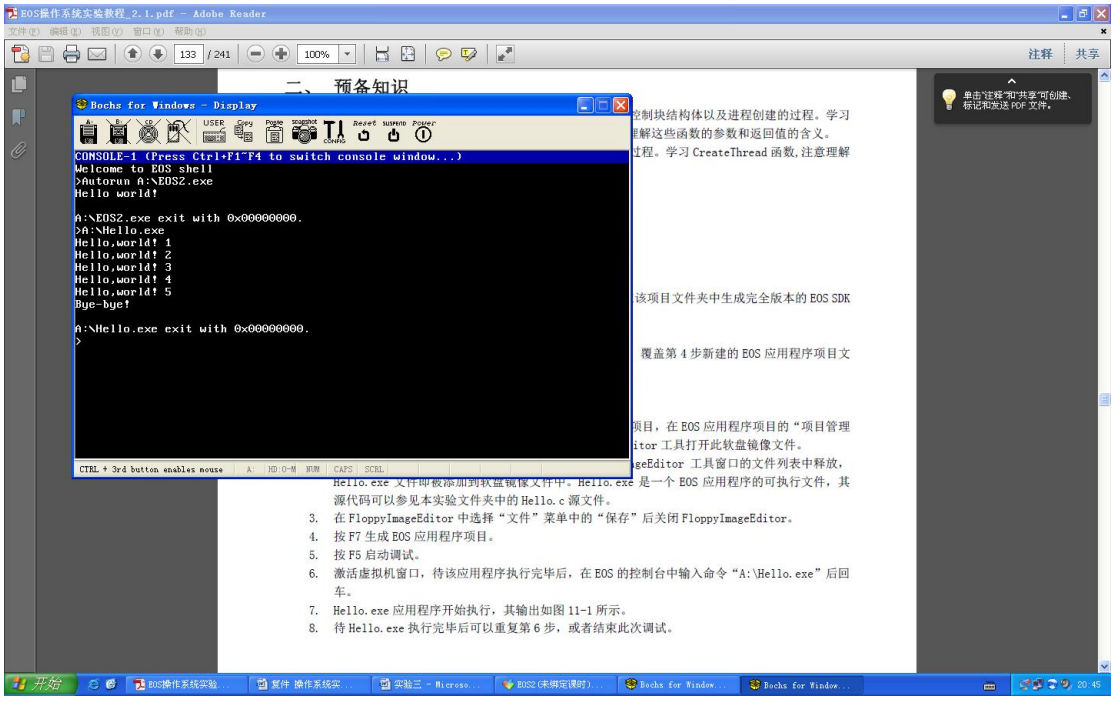
3.1. 准备实验

启动 OS Lab，新建一个 EOS Kernel 项目，分别使用 Debug 配置和 Release 配置生成此项目，从而在该项目文件夹中生成完全版本的 EOS SDK 文件夹，新建一个 EOS 应用程序项目，使用第 3 步生成的 EOS 内核项目文件夹中的 SDK 文件夹，覆盖第 4 步新建的 EOS 应用程序项目文件夹中的 SDK 文件夹。



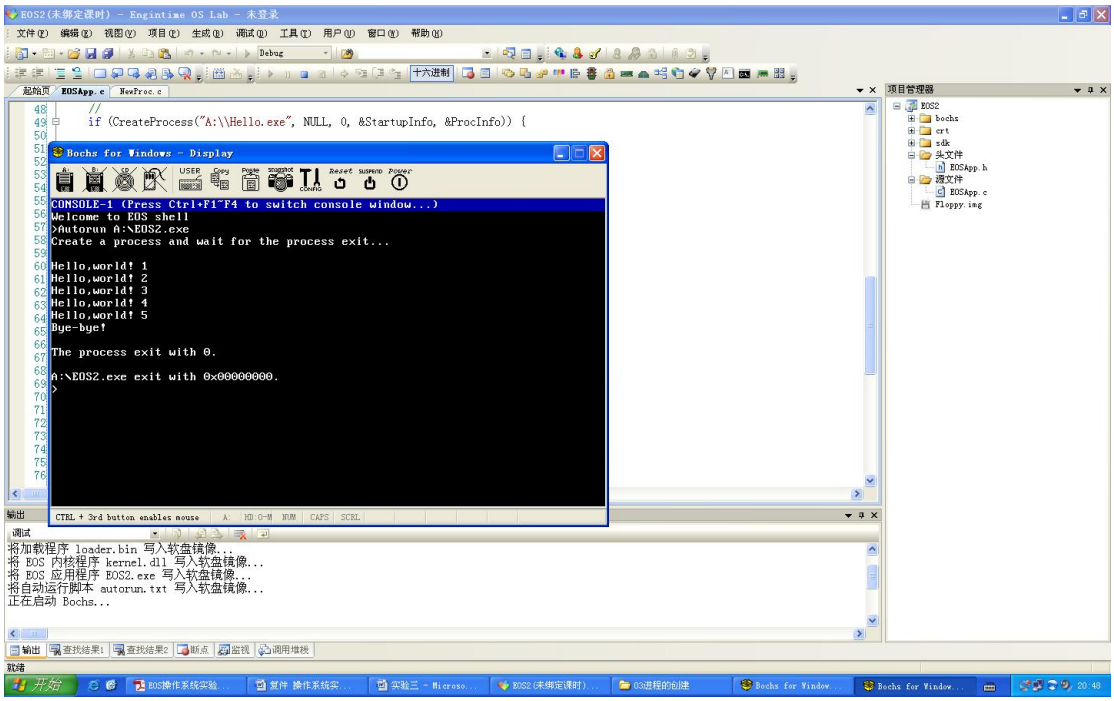
3.2. 练习使用控制台命令创建一个 EOS 应用程序的进程

在 EOS 应用程序项目的“项目管理器”窗口中双击 Floppy.img 文件，使用 FloppyImageEditor 工具打开此软盘镜像文件。将本实验 Hello.exe 拖动到 FloppyImageEditor 窗口释放并保存，生成 EOS 应用程序项目然后启动调试，在 EOS 的控制台中输入命令“A:\Hello.exe”后回车。



3.3. 练习通过编程的方式让应用程序创建另一个应用程序的进程

使用 NewProc.c 文件中的源代码替换之前创建的 EOS 应用程序项目中 EOSApp.c 文件的源代码，重新生成项目然后启动调试，在 EOS 的控制台中输入命令“A:\Hello.exe”后回车。



3.4. 从应用程序的角度理解进程的创建过程

打开 EOS 应用程序 EOSApp.c 文件，在 main 函数中调用 WaitForSingleObject 函数的代码行（第 54 行）添加一个断点。按 F5 启动调试，会在刚刚添加的断点处中断。选择“调试”菜单“窗口”中的“进程线程”菜单，打开“进程线程”窗口，点击此窗口工具栏上的“刷新”按钮。

当前系统中有三个进程，其中第一个是系统进程，镜像名称为空；另外两个是刚刚创建的应用程序的进程，镜像名称分别为“A:\EOS2.exe”“A:\Hello.exe”

数据源: POBJECT_TYPE PspProcessType、POBJECT_TYPE PspThreadType

源文件: ps\psobject.c

进程列表

序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
1	1	Y	24	6	2	"N/A"

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	2	Y	0	Ready (1)	1	0x80017e40 KiSystemProcessRoutine
2	17	Y	24	Waiting (3)	1	0x80015724 IopConsoleDispatchThread
3	18	Y	24	Waiting (3)	1	0x80017f4b KiShellThread
4	19	Y	24	Waiting (3)	1	0x80017f4b KiShellThread
5	20	Y	24	Waiting (3)	1	0x80017f4b KiShellThread
6	21	Y	24	Waiting (3)	1	0x80017f4b KiShellThread

进程列表

序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
2	24	N	8	1	26	"A:/EOS2.exe"

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	26	N	8	Running (2)	24	0x8001f97e PspProcessStartup ← PspCurrentThread

进程列表

序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
3	27	N	8	1	29	"A:/Hello.exe"

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	29	N	8	Ready (1)	27	0x8001f97e PspProcessStartup

选择“调试”菜单“窗口”中的“进程控制块”菜单打开“进程控制块”窗口，在此窗口工具栏上的组合框中选择“进程控制块 PID=24”的选项，可以查看 PID 为 24 的进程的详细信息。其中，阻塞在该进程的线程链表中，可以看到 ID 为 18 的控制台线程阻塞在该进程上。

数据源: POBJECT_TYPE PspProcessType

源文件: ps\psobject.c

进程基本信息

进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
24	N	8	1	26	"A:/EOS2.exe"

进程地址空间(Pas)

进程地址空间的开始虚页号	0x10	进程地址空间的结束虚页号	0x7ffef
页目录	0x409	PTE计数器数据库的页框号	0x408

进程的内核对象句柄表(ObjectTable)

HandleTable	0xa0003000	FreeEntryListHead	0x6	HandleCount	0x5

阻塞在该进程上的线程链表(WaitListHead)

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	18	Y	24	Waiting (3)	1	0x80017f4b KiShellThread

按 F10 单步执行 WaitForSingleObject 函数，再次刷新进程线程窗口，确认子进程已经结束运行。

进程列表

序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
1	1	Y	24	6	2	"N/A"

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	2	Y	0	Ready (1)	1	0x80017e40 KiSystemProcessRoutine
2	17	Y	24	Waiting (3)	1	0x80015724 IopConsoleDispatchThread
3	18	Y	24	Waiting (3)	1	0x80017f4b KiShellThread
4	19	Y	24	Waiting (3)	1	0x80017f4b KiShellThread
5	20	Y	24	Waiting (3)	1	0x80017f4b KiShellThread
6	21	Y	24	Waiting (3)	1	0x80017f4b KiShellThread

进程列表

序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
2	24	N	8	1	26	"A:/EOS2.exe"

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	26	N	8	Running (2)	24	0x8001f97e PspProcessStartup ← PspCurrentThread

进程列表

序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
3	27	N	8	0	0	"N/A"

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	29	N	8	Terminated (4)	27	0x8001f97e PspProcessStartup

3.5. 从内核的角度理解进程的创建过程

3.5.1. 调试 CreateProcess 函数

打开 EOS 应用程序的 EOSApp.c 文件，在 main 函数中调用 CreateProcess 函数的代码行（第 49 行）添加一个断点。按 F5 启动调试，会在刚刚添加的断点处中断。选择“调试”菜单“窗口”中的“进程线程”菜单打开“进程线程”窗口，点击工具栏上的“刷新”按钮，会显示如下图所示的内容。可以看到当前系统中包含两个进程，一个是系统进程，创建了 6 个线程，另一个是应用程序进程，创建了一个线程，并处于运行态，也就

是在其准备调用 CreateProcess 函数时命中了断点。

进程列表

序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
1	1	Y	24	6	2	"N/A"

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	2	Y	0	Ready (1)	1	0x80017e40 KiSystemProcessRoutine
2	17	Y	24	Waiting (3)	1	0x80015724 IopConsoleDispatchThread
3	18	Y	24	Waiting (3)	1	0x80017f4b KiShellThread
4	19	Y	24	Waiting (3)	1	0x80017f4b KiShellThread
5	20	Y	24	Waiting (3)	1	0x80017f4b KiShellThread
6	21	Y	24	Waiting (3)	1	0x80017f4b KiShellThread

进程列表

序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
2	24	N	8	1	26	"A:/EOS2.exe"

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	26	N	8	Running (2)	24	0x8001f97e PspProcessStartup ← PspCurrentThread

由于此时是在内核中的 CreateProcess 函数内中断的，所以先来看一下内核在 4G 虚拟地址空间中的位置。选择“调试”菜单“窗口”中的“虚拟地址描述符”菜单项，打开“虚拟地址描述符”窗口，从该窗口的工具栏下拉框中选择“进程控制块 PID=1”选项，此时显示的是系统进程的虚拟地址描述符表。可以看到，已经使用的虚拟地址空间（底色为灰色的表示已用）都大于 0x80000000。

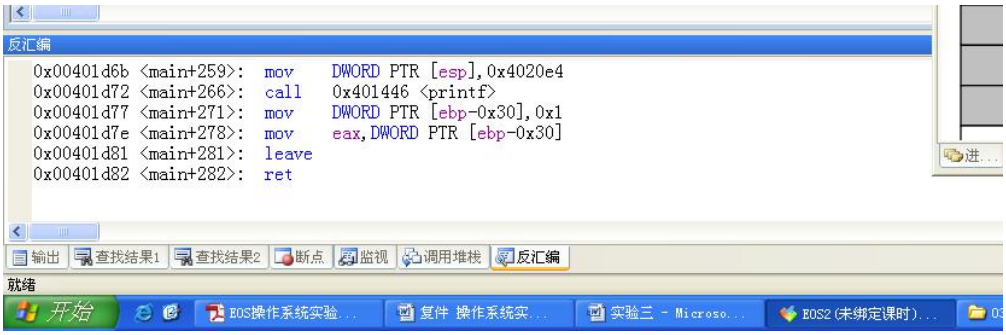
序号	虚拟页框号	虚拟地址
1	0	0x00000000
.....	.....	.....
17	16	0x00010000
.....	.....	.....
524272	524271	0x7fefffff
.....	.....	.....
655361	655360	0xa0000000
655362	655361	0xa0001000
655363	655362	0xa0002000
655364	655363	0xa0003000
655365	655364	0xa0004000
655366	655365	0xa0005000
655367	655366	0xa0006000
655368	655367	0xa0007000
655369	655368	0xa0008000
655370	655369	0xa0009000
655371	655370	0xa000a000
655372	655371	0xa000b000
655373	655372	0xa000c000
655374	655373	0xa000d000
655375	655374	0xa000e000
655376	655375	0xa000f000
655377	655376	0xa0010000
.....	.....	.....
1048576	1048575	0x3ffff000

在“调试”菜单“窗口”中选择“反汇编”，会在“反汇编”窗口中显示 CreateProcess 函数的指令对应的反汇编代码。在“反汇编”窗口的左侧显示的是指令所在的虚拟地址，可以看到，CreateProcess 函数中所有指令的虚拟地址都是大于 0x80000000 的，从而验证了内核程序 kernel.dll）位于高 2G 虚拟地址空间。

接下来确认应用程序在 4G 虚拟地址空间中的位置。在“虚拟地址描述符”窗口工具栏的下拉列表中选择“进程控制块 PID=24”的选项，此时显示的是应用程序进程的虚拟地址描述符表，如下图，可以看到已经使用的虚拟地址都小于 0x80000000。

序号	虚拟页框号	虚拟地址
1	0	0x00000000
.....	.....	.....
17	16	0x00010000
18	17	0x00011000
.....	.....	.....
1023	1022	0x003fe000
1024	1023	0x003ff000
1025	1024	0x00400000
1026	1025	0x00401000
1027	1026	0x00402000
1028	1027	0x00403000
1029	1028	0x00404000
1030	1029	0x00405000
1031	1030	0x00406000
1032	1031	0x00407000
1033	1032	0x00408000
1034	1033	0x00409000
1035	1034	0x0040a000
1036	1035	0x0040b000
.....	.....	.....
655361	655360	0xa00000000
.....	.....	.....
657408	657407	0xa07fffff
.....	.....	.....
1048576	1048575	0x3ffff000

在“调用堆栈”窗口中双击 main 函数项，设置 main 函数的调用堆栈帧为激活状态，在“反汇编”窗口中可以查看应用程序 main 函数的反汇编，可以看到 main 函数所有指令的虚拟地址都是小于 0x80000000 的，这就说明应用程序（eosapp.exe）位于低 2G 的虚拟地址空间中。



选择“调试”菜单中的“窗口”中的“物理内存”菜单项，打开“物理内存”窗口，下图所示，可以查看当前系统物理内存的使用情况。

物理页的数量: 8176

物理内存的大小: 8176 * 4096 = 33488896 Byte

零页的数量: 0

空闲页的数量: 7117

已使用页的数量: 1059

物理页框号	页框号数据库项	状态
0x0	0x80100000	BUSY
0x1	0x80100001	BUSY
.....	.....	.....
0x421	0x80100421	BUSY
0x422	0x80100422	BUSY
0x423	0x80100423	FREE
0x424	0x80100424	FREE
.....	.....	.....
0x1fee	0x80101fee	FREE
0x1fef	0x80101fef	FREE

3.5.2. 调试 PsCreateProcess 函数

在 PsCreateProcess 函数中首先调用 PspCreateProcessEnvironment 函数来创建进程控制块，在 PsCreateProcess 函数中找到调用 PspCreateProcessEnvironment 函数的代码行（create.c 文件的第 163 行），并在此行添加一个断点。按 F5 继续调试，到此断点处中断。按 F11 调试进入 PspCreateProcessEnvironment 函数。

在调用 ObCreateObject 函数的代码行（create.c 文件的第 418 行）添加一个断点，按 F5 继续调试，到此断点处中断，按 F10 执行此函数后中断，此时为了查看进程控制块中的信息，选择“调试”菜单“窗口”中的“进程控制块”，打开“进程控制块”窗口，在该窗口工具栏的下拉列表中选择“进程控制块 PID=27”，可以看到该进程控制块中各个成员变量的值。由于目前只是新建了进程控制块，还没有为其中的成员变量赋值，所以值都为 0。

数据源: POBJECT_TYPE PspProcessType

源文件: ps\psobject.c

进程基本信息

进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
27	N	0	0	0	"N/A"

进程地址空间(Pas)

进程地址空间的开始虚页号	0x0	进程地址空间的结束虚页号	0x0
页目录	0x0	PTE计数器数据库的页框号	0x0

进程的内核对象句柄表(ObjectTable)

HandleTable	0x0	FreeEntryListHead	0x0	HandleCount	0x0

阻塞在该进程上的线程链表(WaitListHead)

接下来调试初始化进程控制块中各个成员变量，首先创建进程的地址空间，即 4G 虚拟地址空间。在代码行（create.c 文件的第 437 行）NewProcess->Pas = MmCreateProcessAddressSpace()；添加一个断点，按 F5 继续调试，到此断点处中断，按 F10 执行此行代码后中断。

在“进程控制块”窗口中，选择“进程控制块 PID=27”，可以看到该进程控制块的“进程地址空间”的值已经不再是 0。说明已经初始化了进程的 4G 虚拟地址空间。

进程基本信息

进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
27	N	0	0	0	"N/A"

进程地址空间(Pas)

进程地址空间的开始虚页号	0x10	进程地址空间的结束虚页号	0x7ffef
页目录	0x423	PTE计数器数据库的页框号	0x424

进程的内核对象句柄表(ObjectTable)

HandleTable	0x0	FreeEntryListHead	0x0	HandleCount	0x0

阻塞在该进程上的线程链表(WaitListHead)

使用 F10 一步步调试 PspCreateProcessEnvironment 函数中后面的代码，在调试的过程中根据执行的源代码，可以在“进程控制块”窗口中查看“进程控制块 PID=27”的信息，或者在“监视”窗口中查看*NewProcess 表达式的值，观察进程控制块中哪些成员变量是被哪些代码初始化的，哪些成员变量还没有被初始化。

进程基本信息

进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
27	N	8	0	0	"N/A"

进程地址空间(Pas)

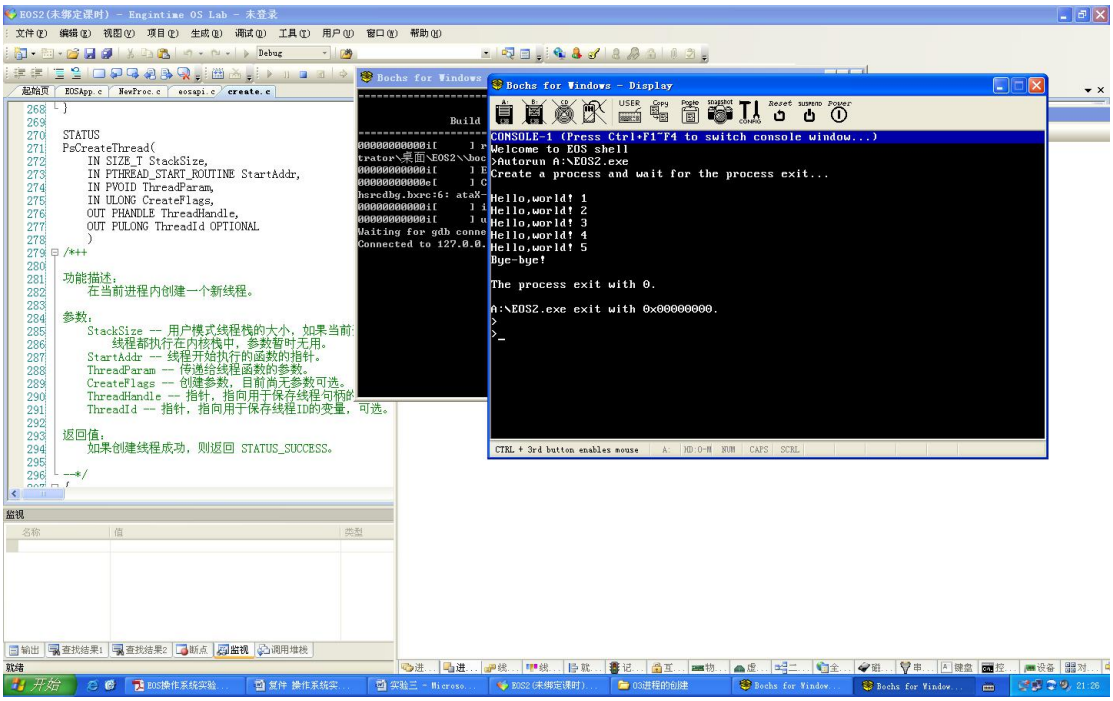
进程地址空间的开始虚页号	0x10	进程地址空间的结束虚页号	0x7ffef
页目录	0x423	PTE计数器数据库的页框号	0x424

进程的内核对象句柄表(ObjectTable)

HandleTable	0xa0004000	FreeEntryListHead	0x1	HandleCount	0x0

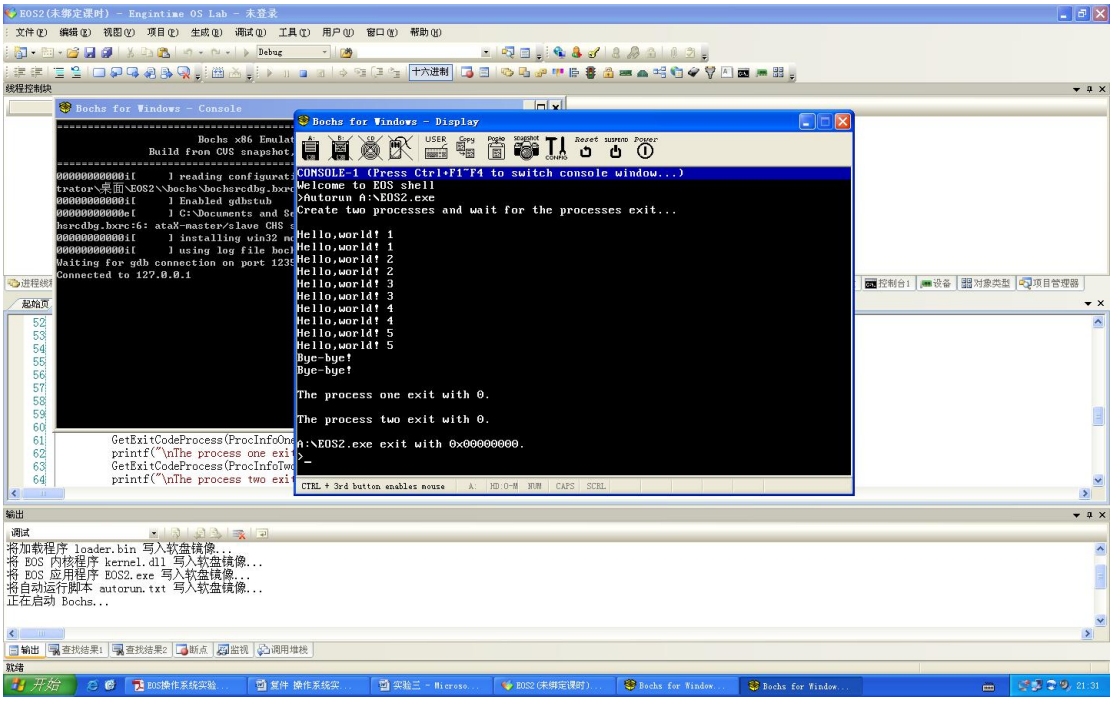
阻塞在该进程上的线程链表(WaitListHead)

按 F5 继续执行，EOS 内核会为刚刚初始化完毕的进程控制块新建一个进程。激活虚拟机窗口查看新建进程执行的结果。



3.6. 练习通过编程的方式创建一个应用程序的多个进程

使用 OS Lab 打开本实验文件夹中的参考源代码文件 NewTwoProc.c，仔细阅读此文件中的源代码。使用 NewTwoProc.c 文件中的源代码替换 EOS 应用程序项目中 EOSApp.c 文件内的源代码，生成后启动调试，查看多个进程并发执行的结果。



3.7. 在应用程序进程中创建一个工作线程（Worker Thread）

使用本实验文件夹中的 AppProg.c 文件中的源代码替换 EOSApp.c 文件内的源代码。删除所有断点后，在工作线程函数 AppThread 中（第 63 行）添加一个断点。按 F5 启动调试，在断点的位置中断。刷新“进程线程”窗口，可以看到下图所示的内容。在应用程序进程的线程列表中，包含两个线程，一个是应用程序的主线程，处于阻塞状态，说明其正在等待工作线程结束；另一个是在应用程序中通过调用 CreateThread 函数创建的工作线程，处于运行状态，说明其正在执行线程函数并命中了断点。

进程列表

序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
2	24	N	8	2	26	"A:/EOSApp.exe"

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)	
1	26	N	8	Waiting (3)	24	0x8001f886 PspProcessStartup	
2	27	N	8	Running (2)	24	0x401d00 AppThread	PspCurrentThread

在“线程控制块”窗口工具栏的组合框中选择“线程控制块 TID=27”的选项，可以查看工作线程的详细信息，如下图所示，包括：线程的基本信息、线程执行在内核状态的上下文环境状态、所在状态队列的链表项（由于该线程处于运行态，也就不在任何状态队列中，所以 Prev 和 Next 指针的值都为 0）、有限等待唤醒的计时器、线程在执行内核代码时绑定的地址空间、阻塞在该线程上的线程链表。其中，阻塞在该线程上的线程链表中显示了应用程序的主线程阻塞在该工作线程上，也就是主线程在等待该工作线程结束。

数据源: POBJECT_TYPE PspThreadType

源文件: ps\psobject.c

线程基本信息

线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名称 (StartAddress And FuncName)	剩余时间片 (RemainderTicks)	
27	N	8	Running (2)	24	0x401d00 AppThread	6	PspCurrentThread

线程执行在内核状态的上下文环境状态(KernelContext)

Eax	0x0	Ebp	0x0	Ebx	0x0	Ecx	0x0
Edi	0x0	Edx	0x0	EFlag	0x200	Eip	0x8001f930
Esi	0x0	Esp	0xa0012ffc	SegCs	0x100008	SegDs	0x77e50010
SegEs	0x77e50010	SegFs	0x77e50010	SegGs	0x77e50010	SegSs	0x77e50010

所在状态队列的链表项(StateListEntry)

Prev	0x0	Next	0x0

有限等待唤醒的计时器(WaitTimer)

IntervalTicks	0x0	ElapsedTicks	0x0
Parameter	0x0	TimerRoutine	0x0

线程在执行内核代码时绑定的进程地址空间(AttachedPas)

进程地址空间的开始虚页号	0x10	进程地址空间的结束虚页号	0x7ffef
页目录	0x409	PTE计数器数据库的页框号	0x408

阻塞在该线程上的线程链表(WaitListHead)

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	26	N	8	Waiting (3)	24	0x8001f97e PspProcessStartup

3.8. 系统线程的创建过程

新建一个 EOS Kernel 项目，打开 ke\start.c 文件，在第 72 行代码 KeThreadSchedule();处添加一个断点，按 F5 启动调试，在断点处中断，刷新“进程线程”窗口，显示如图 11-15 所示的内容。可以看到当前系统中，只创建了一个线程，该线程就是由第 66 行的 PsCreateSystemProcess 函数在创建系统进程后，为系统进程创建的第一个子线程，同时也是系统进程的主线程。

数据源: POBJECT_TYPE PspProcessType、POBJECT_TYPE PspThreadType

源文件: ps\psobject.c

进程列表

序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
1	1	Y	24	1	2	"N/A"

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	2	Y	24	Ready (1)	1	0x80017e40 KiSystemProcessRoutine

打开 ke\sysproc.c 文件，在 KiSystemProcessRoutine 函数中的（第 123 行代码处）添加一个断点，按 F5 继续调试，在断点处中断。刷新“进程线程”窗口，可以看到系统中唯一的线程处于运行状态，说明该线程正在执行其线程函数并命中了断点。

数据源: POBJECT_TYPE PspProcessType、POBJECT_TYPE PspThreadType

源文件: ps\psobject.c

进程列表

序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
1	1	Y	24	1	2	"N/A"

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	2	Y	24	Running (2)	1	0x80017e40 KiSystemProcessRoutine ← PspCurrentThread

当前中断位置处的代码行用于创建一个新的系统线程，并使用该新建线程继续进行操作系统的初始化工作。所以，按 F10 单步调试一次后，刷新“进程线程”窗口，会显示如下图所示的内容，可以看到当前系统中又多出了一个线程，并处于就绪状态。

数据源: POBJECT_TYPE PspProcessType、POBJECT_TYPE PspThreadType

源文件: ps\psobject.c

进程列表

序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
1	1	Y	24	2	2	"N/A"

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	2	Y	24	Running (2)	1	0x80017e40 KiSystemProcessRoutine ← PspCurrentThread
2	3	Y	24	Ready (1)	1	0x80017ed0 KiInitializationThread

打开 ke\sysproc.c 文件，在 KiInitializationThread 函数中（第 160 行代码处）添加一个断点，按 F5 继续调试，在断点处中断，刷新“进程线程”窗口，可以看到系统初始化线程处于运行状态，说明该线程正在执行其线程函数并命中了断点。

数据源: POBJECT_TYPE PspProcessType、POBJECT_TYPE PspThreadType

源文件: ps\psobject.c

进程列表

序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
1	1	Y	24	2	2	"N/A"

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	2	Y	0	Ready (1)	1	0x80017e40 KiSystemProcessRoutine
2	3	Y	24	Running (2)	1	0x80017ed0 KiInitializationThread ← PspCurrentThread

按 F10 单步调试执行 IoInitializeSystem2 函数，完成基本输入输出的初始化。刷新“进程线程”窗口，显示如下图所示的内容，可以看到当前系统中已经创建了控制台派遣线程，并处于就绪状态。

数据源: POBJECT_TYPE PspProcessType、POBJECT_TYPE PspThreadType

源文件: ps\psobject.c

进程列表

序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
1	1	Y	24	3	2	"N/A"

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	2	Y	0	Ready (1)	1	0x80017e40 KiSystemProcessRoutine
2	3	Y	24	Running (2)	1	0x80017ed0 KiInitializationThread ← PspCurrentThread
3	17	Y	24	Ready (1)	1	0x80015724 IopConsoleDispatchThread

由于第 168 行的 for 语句会循环创建四个控制台线程，所以，读者可以在 184 行代码处添加一个断点，按 F5 继续调试。待命中断点后，刷新“进程线程”窗口，会显示如下图所示的内容，可以看到当前系统已经完成了四个控制台线程的创建，并且所有的控制台线程都处于就绪状态。

数据源: POBJECT_TYPE PspProcessType、POBJECT_TYPE PspThreadType

源文件: ps\psobject.c

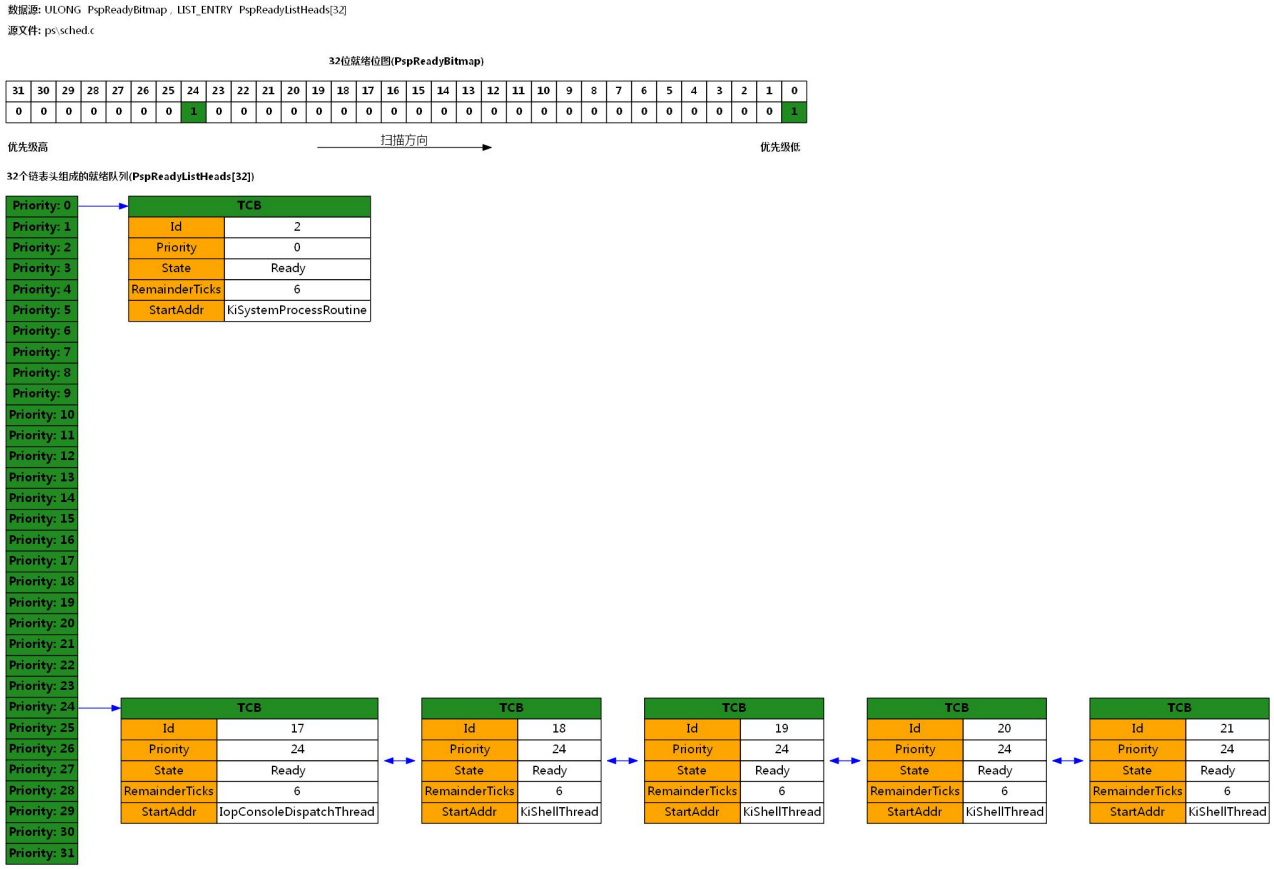
进程列表

序号	进程 ID	系统进程 (System)	优先级 (Priority)	线程数量 (ThreadCount)	主线程ID (PrimaryThreadID)	镜像名称 (ImageName)
1	1	Y	24	7	2	"N/A"

线程列表

序号	线程 ID	系统线程 (System)	优先级 (Priority)	状态 (State)	父进程 ID (ParentProcessID)	起始地址与函数名 (StartAddress And FuncName)
1	2	Y	0	Ready (1)	1	0x80017e40 KiSystemProcessRoutine
2	3	Y	24	Running (2)	1	0x80017ed0 KiInitializationThread ← PspCurrentThread
3	17	Y	24	Ready (1)	1	0x80015724 IopConsoleDispatchThread
4	18	Y	24	Ready (1)	1	0x80017f4b KiShellThread
5	19	Y	24	Ready (1)	1	0x80017f4b KiShellThread
6	20	Y	24	Ready (1)	1	0x80017f4b KiShellThread
7	21	Y	24	Ready (1)	1	0x80017f4b KiShellThread

选择“调试”菜单“窗口”中的“就绪线程队列”，打开“就绪线程队列”窗口。点击此窗口工具栏上的“刷新”按钮，会显示如下图所示的内容。顾名思义，就绪线程队列用于管理那些处于就绪状态的线程控制块，可以看到优先级为 0 的空闲线程，以及优先级为 24 的控制台派遣线程和四个控制台线程分别在其优先级对应的就绪队列中，而当前处于运行态的系统初始化线程（线程 ID 为 3）就不在就绪队列中。



选择“调试”菜单“窗口”中的“线程运行轨迹”，打开“线程运行轨迹”窗口。点击此窗口工具栏上的“刷新”按钮，会显示如下图所示的内容，可以查看这些系统线程的状态转换过程和运行轨迹。注意，系统初始化线程（TID=3）在初始化的过程中会由于等待一些硬件设备的响应，从而频繁进入阻塞状态。

图例： 创建 就绪 运行 阻塞 结束

默认最多显示10个线程的120行数据，可使用本窗口工具栏上的“绘制指定范围的线程”按钮，查看需要显示的内容

Tick	TID=2	TID=3	TID=17	TID=18	TID=19	TID=20	TID=21	函数	行号
0	创建							PspCreateThread	572
0	就绪							PspReadyThread	134
0	运行							PspSelectNextThread	462
1		创建						PspCreateThread	572
1		就绪						PspReadyThread	134
2	就绪							PspSelectNextThread	408
2		运行						PspSelectNextThread	462
3		阻塞						PspWait	230
3	运行							PspSelectNextThread	462
4		就绪						PspReadyThread	134
4	就绪							PspSelectNextThread	408
4		运行						PspSelectNextThread	462
4		阻塞						PspWait	230
4	运行							PspSelectNextThread	462
4		就绪						PspReadyThread	134
4	就绪							PspSelectNextThread	408
4		运行						PspSelectNextThread	462
4		阻塞						PspWait	230
4	运行							PspSelectNextThread	462
54		就绪						PspReadyThread	134
54	就绪							PspSelectNextThread	408
54		运行						PspSelectNextThread	462
54		阻塞						PspWait	230
54	运行							PspSelectNextThread	462
55		就绪						PspReadyThread	134
55	就绪							PspSelectNextThread	408
55		运行						PspSelectNextThread	462
55		阻塞						PspWait	230
55	运行							PspSelectNextThread	462
56		就绪						PspReadyThread	134
56	就绪							PspSelectNextThread	408
56		运行						PspSelectNextThread	462
56		阻塞						PspWait	230
56	运行							PspSelectNextThread	462
57		就绪						PspReadyThread	134
57	就绪							PspSelectNextThread	408
57		运行						PspSelectNextThread	462
58		阻塞						PspWait	230
58	运行							PspSelectNextThread	462
59		就绪						PspReadyThread	134
59	就绪							PspSelectNextThread	408
59		运行						PspSelectNextThread	462
59		阻塞						PspWait	230
59	运行							PspSelectNextThread	462
60		就绪						PspReadyThread	134
60	就绪							PspSelectNextThread	408
60		运行						PspSelectNextThread	462
60		阻塞						PspWait	230
60	运行							PspSelectNextThread	462
61		就绪						PspReadyThread	134
61	就绪							PspSelectNextThread	408
61		运行						PspSelectNextThread	462
62		阻塞						PspWait	230
62	运行							PspSelectNextThread	462
63		就绪						PspReadyThread	134
63	就绪							PspSelectNextThread	408
63		运行						PspSelectNextThread	462
63		阻塞						PspWait	230
63	运行							PspSelectNextThread	462
64		就绪						PspReadyThread	134
64	就绪							PspSelectNextThread	408
64		运行						PspSelectNextThread	462
64		阻塞						PspWait	230
64	运行							PspSelectNextThread	462
66		就绪						PspReadyThread	134
66	就绪							PspSelectNextThread	408
66		运行						PspSelectNextThread	462
66		阻塞						PspWait	230
66	运行							PspSelectNextThread	462
67		就绪						PspReadyThread	134
67	就绪							PspSelectNextThread	408
67		运行						PspSelectNextThread	462
71			创建					PspCreateThread	572
71			就绪					PspReadyThread	134
73				创建				PspCreateThread	572
73				就绪				PspReadyThread	134
74					创建			PspCreateThread	572
74					就绪			PspReadyThread	134
75						创建		PspCreateThread	572
75						就绪		PspReadyThread	134
76							创建	PspCreateThread	572
76							就绪	PspReadyThread	134
Tick	TID=2	TID=3	TID=17	TID=18	TID=19	TID=20	TID=21	函数	行号

4. 实验的思考与问题分析

4.1. 在源代码文件 NewTwoProc.c 提供的源代码基础上进行修改，要求使用 hello.exe 同时创建 10 个进程。提示：可以使用 PROCESS_INFORMATION 类型定义一个有 10 个元素的数组，每一个元素对应一个进程。使用一个循环创建 10 个子进程，然后再使用一个循环等待 10 个子进程结束，得到退出码后关闭句柄。

答：

```
#include"EOSApp.h"
int main(intargc,char*argv[]){
    STARTUPINFO StartupInfo;
    PROCESS_INFORMATION  Pro[10];
//使用 PROCESS_INFORMATION 类型定义一个有 10 个元素的数组，每一个元素对应一个进程
    ULONG ulExitCode; //子进程退出码
    INT nResult=0; //main 函数返回值。0 表示成功，非 0 表示失败
#ifdef_DEBUG
__asm("int $3\n nop");
#endif
    printf("Create ten  processes and wait for the processes exit...\n\n");
    //子进程父进程相同的句柄
    StartupInfo.StdInput=GetStdHandle(STD_INPUT_HANDLE);
    StartupInfo.StdOutput=GetStdHandle(STD_OUTPUT_HANDLE);
    StartupInfo.StdError=GetStdHandle(STD_ERROR_HANDLE);
int p[10],i=0;
//创建十个进程
for(i=0;i<10;i++){
    if(CreateProcess("A:\\Hello.exe",NULL,0,&StartupInfo,&Pro[i])==0)
    //子进程创建成功
    p[i]=0;//标志成功
    else{
    //不成功输出信息
    printf("CreateProcess Failed,Error code:0x%X.\n",GetLastError());
    p[i]=1;//标志失败
    nResult=1;
    }
}
//等待子进程运行结束
for(i=0;i<10;i++){
    if(p[i]==0)
    WaitForSingleObject(Pro[i].ProcessHandle,INFINITE);
}
//得到并输出子进程退出码
```

```
for(i=0;i<10;i++){
    if(p[i]==0){
    GetExitCodeProcess(Pro[i].ProcessHandle,&ulExitCode);
    printf("\nThe process %d exit with %d.\n",i,ulExitCode);
    }
}

for(i=0;i<10;i++){
    if(p[i]==0){
    CloseHandle(Pro[i].ProcessHandle);
    CloseHandle(Pro[i].ThreadHandle);
    }
}
    return nResult;
}
```

4.2. 在 PsCreateProcess 函数中调用了 PspCreateProcessEnvironment 函数后又先后调用了 PspLoadProcessImage 和 PspCreateThread 函数，学习这些函数的主要功能。能够交换这些函数被调用的顺序吗？思考其中的原因。

答：PspCreateProcessEnvironment 的主要功能是创建进程控制块，为进程创建地址空间和分配句柄表。PspLoadProcessImage 将进程的可执行映像加载到进程的地址空间中。PspCreateThread 创建进程的主线程。这三个函数被调用的顺序不能改变，首先要为进程创建了地址空间，然后才能加载可执行映像到相应位置，之后主线程根据已经加载了可执行映像选择开始执行位置和指令。

5. 总结和感想体会

通过这次实验，对于操作系统中进程和线程的概念有了更深的了解。在创建进程的实验过程中，可以很直观的通过图表方式观察到进程的状态，明白了进程状态的终止原因。知道了进程创建所需要调用的 CreateProcess 函数，对该函数进行跟踪调试，知道了它的运行过程，了解了管理进程的进程控制块（PCB）。在线程实验中，学会了利用 CreateThread 函数进行线程的创建，明白了线程各个状态转换以及挂起，理解了线程调度的含义。